# Python and Its Tools

Don Peterson 10 Nov 2010
someonesdad1@gmail.com

Please feel free to email me with corrections or suggestions for this document (use the subject "Your Python and Its Tools document").



## Table of Contents

# Introduction

This document discusses the programming language python (along with some of its add-ons) and some of the reasons why I think it's a good tool for technical folks to have in their back pocket. While no tool fits all situations, I've found python to be an effective Swiss army knife[1] of programming.  About the only time I switch to other tools like C++ or C is when python isn't fast enough.  Some of the key reasons I like python are:

- I find I develop working code anywhere from 2 to 10 times faster than doing the equivalent task in C++.
- It encourages readable and understandable code.
- It scales well to big problems.
- It "includes the batteries".  This means it comes with a powerful library of extra stuff.
- Chances are, someone has already solved the problem you're interested in solving.  The web contains thousands of python solutions, free for the asking.
- If I'm not sure how something works in python, I can start an interactive python session and try it out immediately.

**Who this article is aimed at**:  I assume you're a person with a technical background and you have some experience with one or more programming languages.  Thus, this is not an introduction to programming (you can find many of those on the web and at the bookstore).  Some of the topics I discuss might not seem of much immediate use; however, I encourage you to extract the key ideas from the material and perhaps some day spend a little time investigating the topic, as it's likely to pay off some dividends once you get over the hurdle of learning the technique.  That's assuming, of course, that I do a reasonable job of explaining it.  ☺

Roughly half of the material covers the features of the python language and about a quarter discusses the add-on libraries.  The remaining quarter of the material covers examples.

I also will give examples of code that solve fairly elementary problems.  One of my pet peeves is with people who try to explain a topic, then illustrate the solution with some esoteric problem in their technical specialty that I can't understand (perhaps they're trying to show off their erudition). Unfortunately, all they do is drive most people away.  I hope my examples don't do that to you.

References are given as [xx] and refer to the References section.  Words that mean something special to python (e.g., an important concept or a keyword) are shown like **this**.  Links to other sections in the document are shown like **this**.

## Getting started

The first step is to get the basic python distribution installed on your computer.  You first need to choose which version to install, so read the section **Versions** on page 4.

1 Contrast this to perl, which Henry Spencer once described as a "Swiss Army chainsaw".  This is an apt description and is what led me to discover python in 1998, after becoming disappointed with perl and its syntax.

The usual way is to go to http://www.python.org/download/ and download a suitable python installation. Install it on your computer. For Windows, this is simple -- just download the appropriate exe file and run it. For Linux, you can run the app installer program. However, it's likely that python is already installed on your Linux box. For the Mac, I'll assume you can utilize the Mac-related stuff on the python website.

There are also commercial packages containing python that might suit your needs. Enthought and ActiveState are two examples. As far as I can tell (not having used their products), these folks don't provide any python stuff that you can't download yourself from the web. However, at one time Enthought provided a free beta copy of their python package and I tried it out. It was convenient, as it came with lots of the add-ons (such as numpy, matplotlib, etc.) and they all worked together out of the box with a one-click installation. You can also purchase support from these folks.

## Python's features

This section will try to touch on some of the key things about python. Of course, there's too much to cover to claim completeness, so I'm just trying to give a flavor of what python is like to an experienced programmer. The best executive summary of python is probably that of the python website; I've borrowed some of its content in the next paragraph or two.

Python is a high-level dynamic interpreted programming language that is used in many domains and has been compared to Tcl, Perl, Ruby, Scheme, and Java. Some of python's key features are:

- Clear, readable syntax
- Strong introspection capabilities
- Intuitive object orientation
- Natural expression of procedural code
- Full modularity, supporting hierarchical packages
- Exception-based error handling
- High level dynamic data types
- Extensive standard libraries and third party modules for virtually every task
- Extensions and modules easily written in C, C++ (or Java for Jython, or .NET languages for IronPython)
- Embeddable within applications as a scripting interface
- Advanced language features like metaclasses, duck typing, context managers, and decorators.
- Runs on virtually any major platform.
- Is free to use and distribute, even for commercial applications.
- Lots of documentation: built-in, packed with python, web-based, and published books.

One of the best places to learn python is the tutorial that comes with python -- it's excellent. Run an interpreter window next to the tutorial and type in some of the examples to see how things work for yourself.

One thing that surprises many newcomers to python is that indentation matters. Indentation is used to tell python when a block of code is finished. This occasionally angers a few programmers coming from free-form languages like C, but they typically get over the surprise quickly and realize that it makes programs more readable.

Most python code is very readable. This is important, as code is often read much more than it is written. There's even a document that recommends a standard style to use: http://www.python.org/dev/peps/pep-0008. I recommend you follow its recommendations, as they have evolved to yield the most readable code.

An adjective you'll see below is **pure-python**. This is used to describe code (usually, an add-on library) that is written in python only. There are no added modules of e.g. C code that need to be compiled with the tool. This is handy because you can just drop the code into your python

distribution and start using it immediately. I tend to prefer pure-python solutions because they are (usually) portable. However, the tradeoff is that the performance may be less than a solution that contains some of the heavily-executed code in C.

Strategically, I turn to python as my programming language of choice for two key reasons. First, I like the design of the language -- it's a pleasure to use. Second, I can develop (and understand later!) useful programs with python more quickly than any other programming language I've used.

## Versions

One of the first things a person new to python has to do is to choose between using the latest 2.x version and the latest 3.x version. Python 3 is designed to clean up some of the annoying, inconsistent, or poorly-designed features of the language and marks a break where older code may no longer work. The latest version in the 2.x family is 2.7.

Before you think that you must use the latest and greatest 3.x version, I caution you to learn more about the decision. The python website has a page devoted to this topic: http://wiki.python.org/moin/Python2orPython3 and I recommend you peruse it.

I'll give you my take on the situation. I would love to use python 3, primarily for the reason that an integer divided by an integer results, in general, in a float -- this is the right thing to do. In all python versions before python 3, integer division resulted in an integer[2], just like you'd expect in C. This was fine from a programmer's perspective, but it turned out to be a poor choice from most other perspectives. If you and I go to lunch and want to split the $9 bill, we want our arithmetic to get $4.5, not $4. Where this causes lots of pain is in mathematical calculations: two integers get used in a division and result in zero. It can sometimes be a bit of work to track down where the problem occurred. Compare these two lines of code:

```
y = x/3
y = x/3.
```

The first one will result in y being an integer if x is an integer. If 0 < x < 3, then y will evaluate to zero. In contrast, the second line will result in y being the expected floating point value (as long as x isn't an integer that's too large, in which case you'll get an OverflowError exception).

Python 3 changes things so that `y = x/3` behaves as in the second line above. If you really do want integer division, you can specify it by using the double slash as in `x//3`.

Aside: because of this behavior, for all my scripts using python 2.x that might do numerical calculations, I include a line at the top of the script:

```
from __future__ import division
```

This gives a python version 2.x script the python 3 division. You can also get it with a `-Q` command line option (read the documentation), but I prefer not having to remember to supply that.

I have not yet switched to python 3 (I'm still using python 2.6.5). The reason is that I use a number of libraries and I don't want to switch until the libraries I use are all known to work with python 3. If you're just starting out with python, I would recommend using the latest 2.x version because you're more likely to get libraries that support version 2. You can upgrade to python 3 anytime you want (as well as use them side-by-side).

## Documentation

I have become very fond of the Windows help file that comes with the Windows version of python -- it's `Doc/python265.chm` in my python directory. I like it because it contains all the python documentation and it is hyperlinked so that I can find things quickly. When I'm on Linux, I use the

2 Actually, it's something called floor division and it can be even more surprising for people when they divide two integers when one of them is negative. It doesn't work as you'd expect if you're a C programmer. In C, 3/8 == -3/8 == 3/-8 == 0. In python 2.x, -3/8 == 3/-8 == -1. In python 3, -3/8 == 3/-8 == -0.375 and 3/8 == 0.375.

ChmSee application, which can read these Windows help files. You can of course find HTML documentation and use that if it suits your purposes.

The majority of the time I just click on either `Global Module Index` or `General Index` because I usually know what I'm looking for. The module index gets you to a module's documentation and the `General Index` lets you look up a python term. However, if you're new to python, don't gloss over the other major items. The `Library Reference` also links to the module information, but it is organized by functionality, so look there when you're trying to figure out how to do something. The language reference is excellent and isn't just for the computer jocks. Go there to learn exactly how some feature of the language works.

For the complete overview of python, consult the `Python Documentation Contents`. This will give you lots of reading, but you'll find many things of interest.

## Programming paradigms

Python supports multiple programming paradigms. These are styles/methods of programming. Probably the most familiar is the procedural paradigm. This is usually code that executes one line after another and perhaps makes function calls. Languages like Pascal, C, and FORTRAN are procedural languages.

Another paradigm is object-oriented programming. This is also procedural in some sense, but the code is organized into objects. Objects are things that contain both state and methods; a method is a function associated with an object. One benefit of objects is that it enables encapsulation of information (this can improve reuse of code and cause fewer bugs). It also can model the way things work in the real world. For example, a server program for a company's human resources group might be organized around Employee objects, which would contain information about employees and "know how to behave like an employee". Objects can also inherit from other objects. An example could be a Manager object. It would be an Employee object, yet have additional information and capabilities. Yet a function used to calculate the employee's length of service would be able to process either an Employee object or a Manager object -- they'd both have, say, the `LengthOfService()` method. The key idea here is that both of these objects support the same interface.

A third paradigm is functional programming. This is a programming method that tends to use function calls to do all of its work and it avoids keeping information around in variables with its associated side effects. The focus has tended to be academic because of the (theoretical) ability to prove some programs produce correct output. However, this is not practical for most programs typically used. Still, some valuable tools come with the functional programming paradigm and it is worth your time to learn about some of their capabilities.

## Functions

One thing I usually want to know immediately about a programming language is how parameters are passed to a function. For example, in C, all parameters are passed by value[3] (passing pointers simulates passing by reference). In C++, making an argument a reference argument allows true passing by reference (of course, behind the scenes it's just done with pointers).

In python, function parameters are passed by value -- but you need to realize that the value is a **reference** to an object. If the object is **mutable** (capable of being changed, such as a list), then it appears as if the argument was passed by reference because it can be modified in the function. Thus, for example, if you pass in a list or dictionary (data containers discussed below) to a function, the function can modify the object and the calling context will see the changes. In contrast, a string or number are immutable objects and any changes made in the function to them will not be seen by the calling context. Here's some code demonstrating that objects are mutable:

---

3 Which is why C programmers are taught to not pass big structures by value to avoid the time and stack space performance penalties of copying.

```
 1 class X: pass
 2
 3 def ModifyAttribute(x):
 4     x.a = 4
 5
 6 x = X()
 7 x.a = 1.3
 8 print x.a
 9 ModifyAttribute(x)
10 print x.a
```

when run, this code yields

```
1.3
4
```

Line 1 defines a class with no methods or attributes. Line 6 creates an instance of this class. Line 7 adds an attribute[4] named `a` to this class and sets its value to 1.3. The function called in line 9 modifies the object's attribute named `a` and line 10's output proves that it was modified in the function.

If you called `ModifyAttribute()` with an integer or a string, you'd get an exception because these objects do not support attributes named `a`.

## Numbers

Python comes with three types of numbers built in: integers, floating point, and complex numbers. There are actually two types of integers -- the platform's typical integers (e.g., a 32 or 64 bit integer) and long integers, which can be of arbitrary size. This distinction is slowly disappearing and eventually you'll just need to think that all integers are of one type. In fact, in version 2.6 and beyond, that's mostly what you can do today. You'll occasionally see an "L" suffix on a number to indicate that it's a long integer, but that's about all you need to think about in practical code.

Having arbitrary-sized integers means you won't make arithmetic errors. For example, when programming in C, multiplying two integers can often overflow, which means an error if you're not aware of it. Python's long integer capabilities let you work with 1000 digit integers as easily as with 5 digit integers. You don't have to worry about these kinds of arithmetic errors when working with integers in python. If you do wish to e.g. have integers with a fixed number of bits or behave as signed or unsigned integers, it's not hard to write a class that behaves this way.

Floating point numbers are based on the platform's C library's floating point numbers[5]. The `math` module's functions just call into the appropriate C library functions, so floating point math will be familiar (and as aggravating as floating point normally is). There's a `decimal` module that provides arbitrary-precision floating point should you need it; for example, for monetary calculations where the normal floating point isn't good enough. The decimal module is implemented using python's arbitrary-length integers. There are also numerous arbitrary-precision floating point tools on the web; one is `mpmath`, which I'll talk about below.

Complex numbers are composed of two floating point numbers behind the scenes. There's a `cmath` library that provides the usual elementary functions over the complex domain.

The latest versions of python also come with a `fractions` module. This provides support for rational number operations and they interoperate with other python numbers as you'd expect.

---

4 It's called an instance attribute because it is only associated with the instance. Create another instance of the class X and you'll find it doesn't have an attribute named a. A class can have an attribute, but the syntax is to refer to it by e.g., X.a (i.e., using the class name). You'll want to experiment with these attributes to see how they behave.

5 This is for "CPython", probably the most common version of python available. There are other versions of python, such as JPython, which is implemented in Java.

Recent versions of python contain the `numbers` module which defines abstract types like `Complex`, `Real`, `Rational`, and `Integral` that can be used to determine what type of number you have. If you want to know whether something is a number, use `isinstance(x, numbers.Number)`.

## Containers

Python comes with a well-designed set of native containers for data. The containers are **strings**, **lists**, **tuples**, **dictionaries**, and **sets**. Note that the containers are heterogeneous, meaning they can store elements of different types (this excludes strings, which only holds characters, which are technically one character strings).

Their properties are summarized in the following table. In the following, **immutable** means the object cannot be changed.

| Container | Properties |
|---|---|
| string | Immutable. Python strings can be of two types: regular and Unicode. (In python 3, all strings are sequences of Unicode characters.) If you want to append to a string, you can't add characters to an existing string; you have to create a new string, such as `s = s_old + "abc"`. |
| list | The basic, changeable array type (i.e., you can add, remove, and insert elements). Objects of any type can be stored in a list. Lists are denoted by square brackets. Example:<br><br>`mylist = [1, 2, "a string"]` |
| tuple | Immutable. Similar to a list except once the tuple is created, it cannot be modified. Tuples are denoted by parentheses. Example[6]:<br><br>`mytuple = (1, 2, "a string")` |
| dictionary | A mutable container that associates one object with another (also called maps or associative arrays). The object used to find an object in the dictionary is called the key and the associated object is called the value. Only immutable objects are allowed to be keys; the values can be any objects. Dictionaries are denoted by curly braces. Example:<br><br>`mydictionary = {1: "a string", 3: 88.8, 16.3: [1, 2]}` |
| set | A mutable container that allows only one instance of an object in it; this mirrors the behavior of mathematical sets. Example<br><br>`myset = set([1, 2, "a string])`<br><br>There's an immutable form of sets called `frozenset`. |

These containers have a uniform syntax for testing inclusion:

```
if myobject in container:
    do_something()
```

There are other container methods that do the same thing, but the above has become the preferred syntax because it is so readable.

Python's lists have methods that let you immediately use them as a stack (the methods are `append()` and `pop()`). The `collections` module has a `deque` object that can be used to emulate data structures like queues, deques, and circular buffers.

---

6 A fine point in python's grammar is that tuples are actually defined by the commas, not the parentheses. The example line would work fine without the parentheses.

Should you need to build more complicated data structures, these basic container types give you powerful building blocks. For example, many data structures textbooks give implementations for various data structures (e.g., linked lists, trees, heaps, etc.) based on arrays that are accessed through their indices. The books' code can be utilized directly using lists as the arrays because lists support the C-like indexing access such as `mylist[3]`.

**Important**: note that all indexing of arrays in python is 0-based, meaning the indices start at 0. This is the same as the C language.

Python arrays support "slice" notation, something I'm very fond of. The notation `mylist[n:m:s]` describes the elements starting at index `n` and going up to but not including `m`, and doing so in step sizes of `s` (i.e., the included indices are in an arithmetical progression). Array indices are also allowed to be negative; for example, `mylist[-1]` refers to the last element in the list. Subsequences (lists, strings, tuples, and other like-behaving containers are often called sequences) can be described by slice syntax such as `mylist[:4]`, which means the list from elements 0 to 3. Slice notation is compact, powerful, and heavily-used.

### *Unpacking*

Python supports a quite useful syntax called unpacking. If you start using python, you'll probably use this feature a lot because it reduces code length, yet is still readable. Unpacking refers to allowing multiple assignments to variables in one statement:

```
>>> a, b, c = 1, 2, 3
>>> print a, b, c
1 2 3
```

The right hand side of the assignment statement is really a tuple. The tuple's values are "unpacked" into the given variables. A requirement is that the number of variables match the number of items to be unpacked (or be less than, in which case the last variable will hold the "overflow" as a tuple). You'll also see this unpacking pattern used a lot in `for` loops:

```
for i, j in zip(listA, listB):
  do_something(i, j)
```

The `zip` built-in function returns tuples composed of corresponding elements of the arguments. For example, `zip([1, 2], [3, 4])` returns `[(1, 3), (2, 4)]`.

Unpacking leads to a standard python idiom for swapping the values of variables:

```
>>> a, b = 1, 2
>>> print a, b
1 2
>>> a, b = b, a
>>> print a, b
2 1
```

In fact, this unpacking pattern is more general and works with any iterable:

```
a, b, c = 1, 2, 3
print a, b, c
a, b, c = [4, 5, 6]
print a, b, c
a, b, c = set([7, 8, 9])
print a, b, c
a, b, c = iter([10, 11, 12])
print a, b, c
a, b, c = {13:0, 14:0, 15:0}
print a, b, c
def MyGenerator():
    for i in range(16, 19):
        yield i
a, b, c = MyGenerator()
print a, b, c
```

results in

```
1 2 3
4 5 6
8 9 7
10 11 12
13 14 15
16 17 18
```

I've highlighted two of the output lines in red; these are for the set and the dictionary. Both of these containers are unordered, so you will not, in general, get the elements back in the order you put them into the container. Note the line for the set is not ordered the same as the input was. The dictionary's line is, but this is just a fluke (try a different set of keys to see a different order).

The above example uses two features, iterators (`iter()`) and generators, that are discussed below.

## Control flow

Control flow means the statements a language supports that change the procedural flow of the program. Python has three primary keywords used to control program flow: `if`, `while`, and `for`.

`if` statements are used everywhere and are of the form

```
if conditional:
    do something
elif conditional:
    do another thing
else
    do something else
```

Programmers from other languages often get upset that there is no `switch` statement in python. Instead, an `if -- elif -- else` construct can be used to get the same behavior.

There are two loop constructs: `while` and `for`. The `while` loop continues while a conditional is true:

```
while conditional:
    do something
```

The `for` loop iterates over a sequence:

```
for item in container:
    do_something(item)
```

That's it -- that's all there is to control flow. There are details, like `break` and `continue` in `for` and `while` loops (they behave like they do in C), but you have the big picture; all python programs are built with these constructs.

OK, there's one more case that needs to be covered, as it also affects program flow. This is the conditional expression, as shown in the right hand side of this assignment statement:

```
x = 4 if value == 7 else 2
```

This syntax was put into the language by popular demand (probably because of people's familiarity with C's ternary operator). Some people recommend that if you use it, enclose the right hand side in parentheses to make it stand out. One nice benefit of these conditional expressions is that they can be used in anonymous functions, even though a normal `if` statement cannot. I find I often use them in return statements to replace code like

```
if value >= 0:
    return mylist
else:
    return []
```

with something shorter:

```
return mylist if value >= 0 else []
```

Technically, program flow can also be changed by exceptions. We'll look at those next.

## Exceptions

Unexpected things (exceptions) happen when writing programs. These can be caused by a variety of things, such as a division by zero, a keyboard interrupt, a programming error, or an IO timeout.

Many modern programming languages provide exception handling to deal with these exceptions. Python lets you catch exceptions and respond as you choose. A `try` block is used to do this. Here's an example:

```
try:
    velocity = distance/time
except ZeroDivisionError:
    print "Time is zero"
else:
    print "Velocity is", velocity
```

The statement(s) after the `try` block are run and if an exception occurs, the interpreter looks for an exception handler to handle the exception that occurred. In this case, if `time` is 0, we'll get a division by zero error and the user will get a message that the time is zero. Otherwise, if the velocity calculates without an error, the `else` statement is executed and the user sees the velocity printed.

Other errors could occur in the line that calculates the velocity. For example, if `distance` was a string, a `TypeError` exception would be raised (or thrown, if you like that term). Our example code doesn't handle that exception; the interpreter would then propagate that exception up the call chain until an appropriate handler was found. If no handler was found, program execution halts and a stack traceback message is printed. For example, here's what happens if you try to divide a string by an integer in an interactive session:

```
>>> "1"/3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

When you get a traceback from a running script, you'll get a helpful printout (**backtrace**) of the lines showing you the lines where the error occurred. An experienced python programmer can often figure out what went wrong by looking at the backtrace.

Here's an important point. You'll occasionally see an exception handler that is the bare keyword `except` (i.e., with nothing after it except a colon character). This is syntactically allowed, but is virtually always a bad idea. You see, it will catch every exception -- and this includes things like keyboard interrupts and program errors! You'll probably do this some time and find that it can be quite difficult to figure out why your program isn't working correctly, especially if the error is buried deep in a chain of function calls[7]. I can imagine a few situations where it might be useful (for example, a server process that has to be kept running), but at the very least put in a blatant message somewhere that alerts you that the code under the bare `except` is getting executed.

Exceptions are class objects and you're encouraged to derive your program's exceptions from the base python ones. A common pattern for a program (suppose it is a calculator) is:

```
class CalculatorException(Exception): pass
class BadNumber(CalculatorException): pass
class DomainError(CalculatorException): pass
```

Your program can thus define its own exception hierarchy. Then when an exception occurs, you can write

```
msg = "'%g' is a bad value for the sine function" % value
```

7 This, uh, happened to a friend of mine. ☺

```
raise DomainError(msg)
```

## Modules and libraries

Modules are files containing python code and data[8]. Most of the standard library, for example, is implemented in modules. You access the code in a module by `import`ing it into your script. For example, to use regular expressions in the `re` module, you could include a line like the following in your script

```
import re
```

The data and methods in the `re` module are now available to your script. For example, you might compile a regular expression by the line

```
re_integer = re.compile(r"^[+-]?\d+$")
```

Note the use of the name `re` before the `compile` function call. This is the name of the imported module and it must be used[9]. This helps avoid namespace pollution, which you may have run into if you've written large programs in a language like C. (Once compiled, this regular expression object can be used for matching and replacement farther down in the program.)

You can use modules yourself to collect the code you've written into meaningful groupings. This encourages reuse.

Two things lead to python's power: good design of the core language and the standard library, which contains a huge amount of functionality. I can't begin to cover the library's contents here, as there's just too much stuff. I recommend you study the Library Reference document -- it is organized by functionality and will point you to the modules that interest you. However, at the risk of being thought incomplete by experienced python programmers, I'll give a list of the python modules that I find most useful in my own programs:

```
collections
decimal
functools
getopt
glob
itertools
math
operator
os
os.path
random
re
StringIO
sys
time
```

Note: I use many other modules, but these are the ones I use most frequently.

## Performance

Programmers that are used to using compiled languages worry that the performance of an interpreted language like python will be poor. Of course, in the general case, the worry is valid since it's rare for an interpreted language to be faster than a compiled language. However, here are some thoughts about performance. Also see the comment [ds] in the **References** section.

First, modern computers are quite fast and an interpreted language may be entirely capable of giving you the performance you need. When you factor in the fact that the program will probably be written and working 2 to 10 times faster than the compiled language, the speed benefit of the compiled language might not be so important. And computers keep getting faster. To illustrate this, I have a python script I wrote in 2000 that took over two hours to run on my laptop computer that was the current generation of hardware at the time. Ten years later that identical program ran in 12 minutes on a computer that cost four times less. That's a performance improvement of an order of magnitude.

---

8 There are also modules written in C that are either dynamically loaded or are linked with the python interpreter. This will explain why you don't see some modules in a file like math.py.

9 There are other ways to import modules and the objects inside of modules, but this is a syntactical detail I won't cover here.

Second, numerous things can be done to help make an interpreted program like python run faster. There are profilers, idioms to use to help make your python code run more in the C code of the interpreter (e.g., using iteration tools from the `itertools` module rather than a python `for` loop), and add-ons like psyco.

Third, once you know you do have a performance problem, the best solution might be to write an extension module in C/C++ to fix the area where you know the bottleneck is.

Now, in all honesty, in 12 years of using python, I've had to rewrite code in e.g. C/C++ very few times to get acceptable performance (I can probably count the number of times I needed this on one hand). The only time I've used a profiler is to experiment with it. This doesn't mean that you won't have performance issues; it just means that for the problems I've chosen to work on, python has met my performance needs handily. The reduction in development time virtually always trumps any waiting on the computer -- and that waiting is pretty rare.

## Embedding and extending

Python is written to allow you to embed it in another program. This is a powerful idea, as it lets you e.g. add a scripting language to the other program. Or, the other program can have some of its functionality written in python. Since it's interpreted at runtime, this can let you architect a solution that can be modified/extended by users, even if the program itself is in a compiled form.

Another important feature of python is that it can be extended with non-python code. A common use case is where a python feature or module is implemented in C/C++ code for performance. The python implementation has well-defined interfaces between this extension code and the python interpreter and its environment. Consult the Extending and Embedding section in the python documentation for more details.

# Other features

There are some powerful things out there in the python language and library just waiting for you. I'll discuss a few of my favorites. To get more experienced in python, open the documentation and explore what the various library modules do. Read a lot of other people's code to find out how they do things and what tools they use. Explore how the modules of python's standard library are written and read their test code in the `Lib/test` directory of your python installation.

One of the major software ideas of the last few decades is the idea of a pattern. The word pattern is used to mean that many problems exhibit the same key characteristics when you view them with the proper level of abstraction. This is a well-known phenomenon in mathematics (and, of course, was known long before computers and software appeared). For example, laws for numbers like the associative law $(a+b)+c=a+(b+c)$ also hold for other things more general than numbers; for example, vectors in a vector space or group composition in an algebraic group[10]. In computer science, it has lead to thoughts like those in the book *Design Patterns: Elements of Reusable Object-Oriented Software* [GoF]. Software projects often implement these patterns and then use them in numerous areas.

Python has taken advantage of these patterns in a number of areas. A number of the topics in the following sections derive from patterns that were recognized as a good addition to the language.

By the way, never forget that floating point arithmetic can be non-associative in general. If you don't believe this, see what

```
a, b, c = 0.1, 0.2, 0.3
print a + (b + c) == (a + b) + c
```

10 I've read that while at HP Labs, Alexander Stepanov recognized that many computational tasks are equivalent to various algebraic operations over a suitable algebraic structure and this influenced the design of the C++ STL. If this resonates with you, check out the interesting Algebraic Structure Categories chapter in Barton & Nackman's *Scientific and Engineering C++*, Addison-Wesley, 1994.

```
print a*(b*c) == (a*b)*c
```

prints out. Check it out in other languages too.

## Duck typing

Python doesn't require you to declare the type of a variable before using it[11]. A variable's type is determined by what object it is assigned to (i.e., what object it references). For example, the assignment

```
x = 3
```

makes x of type integer. If the next line is

```
x = "Hello"
```

then x's type was changed to a string.

There are two important ideas common amongst python programmers: duck typing and EAFP. The concept of duck typing is built around the statement "If it walks like a duck, swims like a duck, and quacks like a duck, I call it a duck". This means that often the right thing to do is, if you need a duck object, to assume the object you have is a duck and treat it as a duck. In other words, you make the method calls you'd expect the duck to have. If, for example, the `quack()` and `swim()` methods completed successfully (i.e., no exception was raised), you'd call it a duck even if it was actually a parrot[12].

Here's a more concrete example. Suppose you need to iterate over the contents of an object. You don't care whether it's a container or an iterator or something else that supports the iterator interface. You put it in a `for` loop and if it works, it was the proper type. That's duck typing. You saw an example of this above in the **Unpacking** section.

EAFP means "easier to ask for forgiveness than permission". It describes the behavior described above -- go ahead and assume you have the object that you need and deal with any problems encountered after trying. The reason this can be a good idea is it may make your code simpler. Here are two examples contrasting the approaches. In each case the function is to convert its first parameter to a float and call the second parameter, a univariate function, with the float, returning the result.

```
def eafp_func(x, f):
  return f(float(x))

def lbyl_func(x, f):
    assert isinstance(x, int) or isinstance(x, float)
    assert f is not None
    return f(float(x))
```

`lbyl` means "look before you leap". In the second function, we spent effort trying to qualify that the parameters were proper.

Now, I'm not saying that the second function is the wrong way. I sometimes use that pattern when writing functions. However, the first function is easy to read and understand what it's doing. To understand the second function, you have to read past the `asserts`. Of course, in either case, if something is wrong, an exception will be raised and someone will have to deal with it. But if this function was deep in some numerical code, it might make more sense to let the main calling context deal with the exception (the backtrace will still lead you to the offending line). For example, it might have come from a user's input and the program can infer which parameter was wrong and requery

11 This is called dynamic typing and is contrasted to static typing seen in languages such as C and C++. One criticism of dynamic typing is that a type error might not be caught until runtime and it brings the program to a halt. With static typing, the compiler catches the problem.

12 A veiled reference to the dead parrot sketch of Monty Python; such references are *de rigueur* for python programmers.

the user for the correct input.

Here's a more subtle point. Note that the `lbyl_func` forgot to check to see if `x` is a string, which is certainly allowable because strings can be converted to floats. Down the road, `x` might be another object that can be converted to a float, such as a python `decimal.Decimal` object. Oops, we now have another bug and have to change the assert again. The first solution using duck typing just works as long as we throw something at it that can be used with `float()`. This is demonstrated in the `frange.py` module included in the zip file -- that module can be used many years later with numerical objects that didn't even exist when `frange.py` was written, as long as they support the proper semantics. This is the real benefit of duck typing because, in one sense, it's code reuse.

## Testing your software

Writing tests for your software is important to help give you confidence that it's doing what it's supposed to do. Python has two libraries that can help you with testing.

First is `doctest`. This module is interesting because it searches your code for snippets of code that look like interactive sessions and executes the code, checking the results against what actually happened. The reason this is useful is that it allows you to both instruct your users in how to use your code and write test cases at the same time[13]. Here's a simple example.

The string under a function or method name is called the **docstring**. It is conventional to put a one-line summary of what it does, then follow with more explanation of details if needed. This docstring contains the interactive code that shows a user of the function how it works. In the following listing, the docstring tells us that calling `myfunc(3)` at the interactive prompt would return `4`. The doctest finds lines with the `>>>` prefix in the docstring, executes them, then compares the results to the string on the following line and issues an error message if they don't match.

```
def myfunc(a):
    '''Calculate a+1 and return it.

    Here's an example of incrementing 3:

    >>> myfunc(3)
    4
    '''
    return a + 1

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

The code at the bottom runs the test. If this code is run as a script from the command line, the variable `__name__` contains `"__main__"` (in contrast, if this module was imported, `__name__` would not have this value. If you put this code into a script and run it, you'll get no output. That's as expected, as the `doctest` module doesn't print anything out unless there's a problem (call the script with the `-v` option to see a verbose listing of what's going on).

The other module to help with testing is called `unittest`. This is based on a commonly-used testing pattern used in the software industry. It is object-oriented and provides a rich set of tools for performing tests. To give you a flavor it how it works, here's an equivalent test to the one given above; I changed the code below `myfunc`'s code in the above script to read:

```
import unittest

class TestIncrement(unittest.TestCase):
    def test_three(self):
        self.assertEqual(myfunc(3), 4)
```

13 For a geezer like me, that "user" is usually me a few days or weeks later when I've completely forgotten what was going on.

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
    unittest.main()
```

You derive your test object from `unittest.TestCase`. Your test methods must begin with `test_`. These methods are found by the `unittest` module (through introspection[14]) and run; the results are tallied and printed out. When this script is run, the results printed are:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

Each test case results in a dot being printed to the screen.

Note that I've used both `doctest` and `unittest` in the same file. This is fine if it's appropriate, but usually only one test module is used. To learn lots about the `unittest` module's use, go to the `Lib/test` directory in your python distribution and examine some of the tests python uses to test its library modules.

## Object-oriented programming

An object is a thing with associated state and methods. Objects have been a popular programming paradigm because object orientation is a pattern that can be used to emulate how things behave in the world, thus making it easier to construct solutions to real-world problems.

Python supports object-oriented programming, but doesn't force you to use it exclusively. It supports multiple inheritance. This lets you construct objects that behave in multiple ways. Example: `Employee` could be a base class of all the people in a company. Then various behaviors could be gotten depending on the type of the employee. Suppose that a company required all of its department heads to also be trained to be trainers.

```
class Employee: pass
class Manager(Employee): pass
class Trainer(Employee): pass
class DepartmentHead(Trainer, Manager): pass
```

The `Employee` class contains the behavior and data common to all employees. `Manager` and `Trainer` have behavior and data over and above those of an `Employee`. The last line shows how `DepartmentHead` inherits the properties of both a `Trainer` and a `Manager`. Interface is a key. The implementation can change as long as the interface remains the same -- then this doesn't break software that uses the interface. Multiple inheritance can be valuable when it is used to promote code reuse.

## List comprehensions

List comprehensions are purely syntactic sugar, which means they don't add anything new to the language, but make it "sweeter" for the programmer to use. I consider list comprehensions to be one of the most important additions to python, ever. The reason is that they can concisely express one or more conditional loops, usually on one line. They make your code shorter and easier to read.

I'd hazard a guess that two of the most common patterns in programming are

```
for item in collection:
    do_something(item)
```

and

14 Introspection is enabled in a language by allowing objects to examine themselves at runtime to determine how they behave (e.g., what attributes and methods they have). The abilities of the doctest module to look at docstrings is a form of introspection.

```
for item in collection:
    if condition is True:
        do_something(item)
```

List comprehensions can collapse these loops into one line each:

```
[do_something(item) for item in collection]
[do_something(item) for item in collection if condition is True]
```

These list comprehensions return a list, so they are usually used to modify a list's elements. For example, if we wanted a list of the squares of the even integers greater than or equal to zero and less than `n`, we'd use the list comprehension

```
integers = [i*i for i in xrange(n) if i and not (i % 2)]
```

A slightly subtle point in the `if` part of the expression is that I just used the "short" form `if i` instead of `if i != 0`. This is the recommended **pythonic** way because it's faster and easier to read. The second part of the conditional is true if the remainder of division by two is zero. List comprehensions are concise yet readable because they mimic set notation used in mathematics:

$$B = \{i^2 | i \in 0, 2, 4, \ldots n-1\}$$

You can use list comprehensions with more than one for loop:

```
[i*j for i in mylist1 for j in mylist2]
```

However, it's not difficult to construct complex list comprehensions that are hard to read. If you do, it's better to change them to ordinary `for` loops to make them easier to read.

Another addition to the python language that adds both performance and readability are **generator expressions**. These are constructs that let you skip the construction of the list. For example, if the above list `integers` was going to be iterated over to get the sum of the integers, it would be more straightforward to use a generator expression in conjunction with the `sum()` built-in:

```
mysum = sum(i*i for i in xrange(n) if i and not (i % 2))
```

This is memory-efficient when n is a large number because the whole list of numbers isn't in memory at one time.

Generator expressions are enclosed in parentheses. The script

```
for i in (x**2 for x in range(10)):
  print i,
```

produces

```
0 1 4 9 16 25 36 49 64 81
```

Python 2.7 and 3 unify things by including set comprehensions and dictionary comprehensions:

```
>>> { n*n for n in range(5) } # Set comprehension
{0, 1, 4, 16, 9}
>>>
>>> { n: n*n for n in range(5) } # Dictionary comprehension
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Note both are denoted by curly braces.

## Collections

Another module, `collections`, provides a handy container: the `deque`. This is a short name for double-ended queue. It is a popular data structure that has a number of uses. Basically, it's optimized for producer-consumer actions with adding and removing objects from either end of the container. It is not intended for random access because retrieving an element from the middle of the list is O(n). Use a list if that's what you need. But lists are inefficient when adding or removing

elements from the end with index 0 because of the need to move elements in memory.

The producer generates something and sticks it on one end of the deque[15]. The consumer pops objects off the other end of the deque. The data structure ensures things get processed in the order the producer puts them on the deque (unless, of course, the consumer wants to change the order). You can use the deque as a queue because `push` and `pop` operations are provided for both ends (this also lets you use the deque as a stack, but that's just as easily done with a list, which provides `append()` and `pop()` methods). Deques also support being initialized to a maximum size. When the container reaches the maximum size, the item from the end opposite to the one pushed on is dropped from the container. Read the documentation on deques and you'll find they have numerous useful features.

Another common programming pattern is to go through a sequence of incoming items and group them by some criterion. For example, suppose you are reading lines from a file where each line contains an algorithm method name and an execution time. You want to group the data by each algorithm and provide statistics on the execution times. You'd use a dictionary to hold the data; the key would be the algorithm name and the values would the lists of execution times. Your code might look something like the following:

```
data = {}    # Dictionary to hold the grouped data
for line in input_data:
    algorithm, time = line.split()  # Split whitespace-separated information
    if algorithm not in data:
        data[algorithm] = []        # Add new dict entry and set to empty list
    else:
        data[algorithm].append(time)  # Add time to existing list
# Summarize the data
```

At the end of this code, the dictionary would contain the subgrouped data, ready for further processing.

This is such a common pattern that python's designers decided to add a new data structure called `defaultdict` to the `collections` module. This data structure allows the above code to be shortened to the very readable form

```
data = collections.defaultdict(list)
for line in input_data:
    algorithm, time = line.split()
    data[algorithm].append(time)
```

The `defaultdict`'s constructor takes a "factory" function that produces the default element when creating an entry for a new key[16]. You can see that this allows the removal of the check to see if the key is not already in the dictionary, which then created the element with an empty list.

## Iterators

I have a fondness for iterators. These are objects that let you iterate over something. Not surprisingly, you iterate a lot when writing programs -- for example, the `for` loop is a typical iteration tool. If you have a list, then the `for` loop can iterate over it directly:

```
for i in [1, 2, 5, 12]:
    print i
```

Pretty straightforward. Behind the scenes, python is creating an iterator for you. An iterator is a "lightweight" tool that lets you access the items in order and one at a time. It is called lightweight because all the objects in the sequence do not have to be read into memory at once. This can be important for big problems -- an example is where you want to process a very large text file. You might run out of memory if you tried to read it all into memory at once; however, it would be very

15 A nice feature of these deques is that their insertions and deletions are thread-safe.

16 Because this pattern is very common, I feel it would be a great addition to the language to include this defaultdict behavior with the basic dict object through an optional keyword argument.

manageable if you just read one line into memory at once, processed it, then got the next line.

Let's look at the two ways this reading of a text file is done since it's such a common thing to do. The first method is to read all the lines in at once into a list, then process the items in the list:

```
lines = open(textfile).readlines()
for line in lines:
    process(line)
```

I use this pattern quite a bit and it's fine if you have the memory to do it. But if you want to write robust code that won't break on that very large file, you can instead write the shorter and more readable

```
for line in file(textfile):
    process(line)
```

These look similar, but they accomplish the same task in different ways. The second way creates an iterator for the file and lets the `for` loop access the lines one at a time. This is the preferred pattern for general purpose code because it won't break on a big file and it's efficient and readable.

Python has a built-in command `iter()` that is used to construct iterators from objects that can be iterated over (the documentation for `iter` will tell you the technical requirements on the objects for this to work). The native data types such as list, tuple, dictionary, set, and string can all be iterated over without any hassle. If you iterate on a dictionary, you get the keys.

The idea of an iterator has been powerful in a number of languages. For example, the designers of the STL (standard template library) for C++ thought about them deeply and you'll find lots of iterators in the STL. Python has the `itertools` module which supplies a number of useful iteration tools. I recommend you study it and its examples; you'll find applications for them in short order.

Here's an example script that finds anagrams using an iterator from `itertools`:

```
import sys, itertools
from words import words

def main():
    for i in itertools.permutations(sys.argv[1], len(sys.argv[1])):
        s = ''.join(i)
        if s in words:
            print s

main()
```

The `words` module is something I quickly made from a `words` file from a Linux box. It contains a dictionary named `words` that looks like the following:

```
words = {
    "aardvark": None,
    "aardvarks": None,
    "aardwolf": None,
    "aardwolves": None,
    "aba": None,
    ...
}
```

I used a dictionary because python builds them using hash tables, which means seeing if a word is in the container is fast. A more "proper" data structure here would be the set, but this words file had been written before sets existed in the python language.

This program takes the word you type in on the command line (`sys.argv[1]`) and feeds it to the `itertools` iterator named `permutations`. The first argument is the iterable to permute and the second argument is the number of items to include in each permutation. Since the letters are the iterable items in the string, a tuple of the letters is returned by the iterator. The `join` command puts them together into a string again. Then each permutation is checked to see if it's in the `words`

dictionary; if it is, it's printed out.

While this isn't a comprehensive anagram program (it wouldn't handle capital letters or input words with apostrophes), it's still pretty impressive that so much work can be done with so little code.

Once you see how easy it is, you'll find you'll use permutations and combinations a lot -- one area I've used them for is brute-force searching for solutions to problems. For example, I wrote a program that lets me find how to make a given stack size of gauge blocks from my gauge block set that is missing a number of blocks[17]. If I didn't have this program, the gauge block set would be much less useful than it is.

I believe the `itertools` module is implemented in C, so its functions are fast. (I say this because I find no `itertools.py` module, but I haven't looked at python's source code.)

## Anonymous functions

Python lets you create anonymous functions. These are simple functions where it doesn't make sense to write a "normal" function, as they are typically used in one spot for a specialized purpose. Python allows this because functions are first-class objects, meaning they can be created, passed around, introspected, etc. like other objects.

Here's an example where an anonymous function is used to print out 0 if a number is odd and 1 if the number is even:

```
f = lambda x: 0 if x % 2 else 1
for num in range(5):
    print num, f(num)
```

The syntax of an anonymous function uses the keyword `lambda`, follow by the argument list (here, there's just one argument) and a colon. An expression must follow.

The result is

```
0 1
1 0
2 1
3 0
4 1
```

Anonymous functions are useful in conjunction with the use of functional programming techniques (see below). However, some people feel they are not worth having because they can always be replaced by a regular function -- and regular functions can have e.g. loops and conditionals in them, which anonymous functions cannot have (but you can use the conditional expression as shown above).

My opinion is that anonymous functions should be use sparingly -- only in cases where they aid readability. Here's an example where the use might make sense. The function `make_linear_xfm(m, b)` is used to return a function which will transform a number with the transformation $x' = mx + b$:

```
def make_linear_xfm(m, b):
  return lambda x: m*x + b
```

Certainly, we could also write a function `LinearXfm(x, m, b)` that did the same thing. But sometimes having a univariate function to perform the transformation is handy -- for example, it can allow the use of functional programming tools. See the section below on functional programming for an example.

Here's how you'd create the same function without using an anonymous function:

```
def make_linear_xfm(m, b):
    def f(x):
```

17 This problem is a well-known NP-complete problem called the subset sum problem.

```
        return m*x + b
    return f
```

This example uses a feature called **nested scopes**. This allows one to have a function defined inside another function. The function `f` has access to the `make_linear_xfm`'s scope; i.e., it can use the values of `m` and `b`.

## Generators

Generators are functions that use the keyword `yield` to make the function behave as an iterator. Let's look at an example. We'll write a generator that provides a sequence of a power of the positive integers less than the integer passed into the generator:

```
def PowerGenerator(n, power):
    for i in xrange(n):
        yield i**power

for i in SquareGenerator(4, 3):
    print i
```

The output of this script is

```
0
1
8
27
```

What happens is interesting. When python sees the `yield` keyword, it treats this function specially. When the function is called, it executes normally until the `yield` is encountered. The indicated value is returned and the state of the function is saved. When the `next()` method of the generator is called, processing resumes in the function immediately after the last `yield` statement encountered (a generator can have more than one `yield` statement). Incidentally, this is how iterators are used behind the scenes: their `next()` methods are called until the `StopIteration` exception is raised. This pattern is used in many python built-ins.

Here's one more example -- a quick and dirty 32-bit random number generator [nr]:

```
def QuickAndDirty(seed=0):
    if seed:
        x = seed
    while True:
        x = (1664525*x + 1013904223) % (2**32)
        yield x/float(2**32)
```

While the `random` module that comes with python is the random number generator you should turn to for normal use, this demonstrates a pure-python implementation you can use for quick and dirty tasks. I assume you'll consult [knuth, vol 2] before making any changes to the generator's constants.

Recent versions of python added more power to generators by allowing you to pass objects back into the generator while it is being used. This allows python to have coroutines, which adds to the capabilities of the language.

## Functional programming tools

Functional programming is a different style of programming from the familiar procedural style. I'll refer you to the references for more details [fp], as it takes a bit of reading and thinking to understand the details. Here, I'll briefly look at some of the functional programming tools in python. You can certainly write your programs without using them, but as you learn their capabilities, you'll find yourself using them more and more. They can provide a lot of power, but they can also make your program harder to read and understand if you over-use them.

The basic functional programming tools in python are `map`, `filter`, and `reduce`. These are

functions that apply other functions to sequences. Here, sequences can be things like tuples and lists, but also things that involve **lazy evaluation** like generators or iterators. I'll use lists in the examples.

`map` causes a function to be applied to each element of the list. Suppose we use the function returned from `make_linear_xfm(2, 3)` given in the section on anonymous functions above and apply it to the list `[1, 2, 3]`:

```
f = make_linear_xfm(2, 3)
print map(f, [1, 2, 3])
```

This will print the list `[5, 7, 9]`.

The `map` call here could be replaced by a list comprehension: `[f(i) for i in [1, 2, 3]]`. I would hazard a guess that most folks (including me) would feel this is easier to read and understand what's going on than `map(f, [1, 2, 3])`.

Basically, `filter` does the same thing as `map` with the addition of only outputting the elements that meet a conditional. `reduce` is used with a binary operator to turn a sequence into a single object (the section Symbolic computation with SymPy on page 30 shows a use of `reduce` to calculate a product. Consult python's documentation for more details.

The consensus amongst python programmers seems to be that list comprehensions are preferred over using these functional programming commands. The reason is that the list comprehensions tend to be more readable. My suggestion is to use functional programming tools where they make sense.

There is, however, one area where the functional programming tools may be preferable: when you're after more performance. Here is a script that compares the timing of the `map` function compared to the list comprehension:

```
from time import time

def make_linear_xfm(m, b):
    return lambda x: m*x + b

n, array, func = 5000, range(10000), make_linear_xfm(2, 3)

# Method 1:  list comprehension
t = time()
for i in xrange(n):
    b = [func(i) for i in array]
tlistcomp = time() - t

# Method 2:  map
t = time()
for i in xrange(n):
    b = map(func, array)
tmap = time() - t

print "list comp:  %.2f s" % tlistcomp
print "map:  %.2f s" % tmap
print "ratio = %.3f" % (tlistcomp/tmap)
```

The results of running this on my computer were:

```
list comp:  16.50 s
map:  13.06 s
ratio = 1.263
```

A conclusion is that a functional programming construct may reduce running time on the order of a fourth. Of course, you'll want to measure things in more detail for real problems, but when you need more performance, this is something to be aware of.

# Math

Python is a quite capable tool for math and technical work with the stuff that's built into the language and the standard library. But there are powerful add-on tools that give you more capabilities.

You can get an overview of tools at http://wiki.python.org/moin/NumericAndScientific. This list might be a bit intimidating at first, so I'd suggest you start with numpy and matplotlib.

When you're attacking a new problem with python, it makes sense to see if others have already solved the problem you're working on (or one closely related). See [res] for more ideas.

## Array processing with numpy

numpy (http://www.scipy.org/) is a popular array processing tool that replaced two earlier array processing tools, Numeric and Numarray. The basic design is to allow you to create arrays based on the number types in the typical C implementation, then let you operate on these arrays with C-based transformation tools. This gives good performance.

The documentation of numpy still leaves something to be desired, although people are slowly working on it to improve it. Still, it's worth learning because it's a powerful tool for scientific computing. I'll give a few examples of its use. In the following, assume we've executed the lines

```
from __future__ import division
from numpy import *
```

(The first line will let me write expressions like 1/8 and get 0.125 for the result rather than 0.)

The basic component of most usage is an array of values. numpy arrays can be of almost any shape, but probably the most common are vectors and matrices. I'll just look at vectors here; matrices are analogous, but require one to learn the details of indexing. One common way of producing a vector is

```
>>> arange(0, 1, 1/8)
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,  0.75 ,  0.875])
```

The `arange` command produces a vector of values that start at 0 and go up to but not including 1 in steps of 0.125. However, since this is implemented with the platform's C library, you will get floating point roundoff errors that sometimes give you one more number than you expect:

```
>>> arange(9.6001, 9.601, 0.0001)
array([ 9.6001,  9.6002,  9.6003,  9.6004,  9.6005,  9.6006,  9.6007,
        9.6008,  9.6009,  9.601 ])
```

This is a flaw with binary floating point arithmetic and is not a flaw with numpy.

Another way to produce a vector is to convert a list:

```
>>> array([1, 2, 3.0])
array([ 1.,  2.,  3.])
```

Conversely, you can convert a numpy array `A` to a list using `list(A)`.

The power of numpy's vectors come from the fact that you can do arithmetic with them and combine them with other arrays. You can iterate on them using the familiar python indexing notation. You can create new arrays with slices. Here's an example of setting all the elements with even indices to zero:

```
>>> a = arange(0, 1, 1/8)
>>> a[0::2] = 0
>>> print a
[ 0.     0.125  0.     0.375  0.     0.625  0.     0.875]
```

Note numpy prints out its arrays in a format that looks like a list, but doesn't include the commas[18].

18 The difference you see between just typing 'a' and 'print a' are caused by the difference between repr(a) and

I can multiply this numpy array by a scalar:

```
>>> 8*a
array([ 0.,  1.,  0.,  3.,  0.,  5.,  0.,  7.])
```

This multiplies each element of the array by the scalar. Multiplying two arrays of the same size causes pairwise multiplication (you'll get an error if you try to multiply two arrays of different sizes):

```
>>> a*a
array([ 0.      ,  0.015625,  0.      ,  0.140625,  0.      ,  0.390625,
        0.      ,  0.765625])
```

This just gives a taste of the things in numpy; there are too many other things to do it justice. However, let me list some of the things that might be of interest:

| | |
|---|---|
| polyfit | Fit a polynomial to data, minimizing the squared error. |
| Elementary functions | Most elementary functions that you would expect are present. |
| Statistics | Calculate numerous elementary statistics and covariances and generate random numbers from a variety of distributions. |
| Discrete Fourier transforms | Calculate FFTs and inverse FFTs in full complex number form. |
| Boolean selection | Select values of arrays using Boolean arrays to indicate which objects to choose. |
| Cumulative products and sums | |
| Bit operations | |
| Polynomial operations | |
| Linear algebra | |
| Set operations | |
| Comparisons | |

Many of the things that can be done easily with numpy can be done with regular arrays of numbers. For example, if you had two lists of numbers, you can multiply them together to get a new list:

```
>>> import operator
>>> map(operator.mul, [1, 2, 3], [4, 5, 6])
[4, 10, 18]
```

You could use such techniques to construct a library to do array arithmetic similar to what numpy does. This might be appropriate if you wanted to use e.g. high precision numeric types in your arrays. However, these pure-python solutions will be much slower than the equivalent operations in numpy, as numpy does them in compiled C libraries. While numpy should meet most needs, it's nice to know python has the flexibility and power to construct special-purpose solutions when needed.

One other comment. Using numpy is fast because it's written in C. Be aware of this when you need a big list of numbers for an application -- you may be able to quickly do it in numpy, then convert it to a list. Here's a demonstration showing that numpy is faster than constructing your own list with pure-python code:

```
import random, numpy as N
from time import time
```

---

str(a); I'll refer you to the python documentation for the difference between these two string representations of an object.

```
n = int(1e6)
print "For %d points:" % n
a = []
t = time()
for i in xrange(n):
    a.append(random.normalvariate(0, 1))
print "  Using random module = %.2g s" % (time() - t)
t = time()
a = N.random.normal(0, 1, 10*n)
print "  Using numpy for 10*n = %.2g s" % (time() - t)
```

with the following results:

```
For 1000000 points:
  Using random module = 2 s
  Using numpy for 10*n = 0.85 s
```

This shows using numpy to get random normal deviates takes (100)0.085/2 or less than 5% of the time to do it with a python library tool. Most of the time is in the call to `random.normalvariate()`; adding it to the array only takes about 10% of the total time.

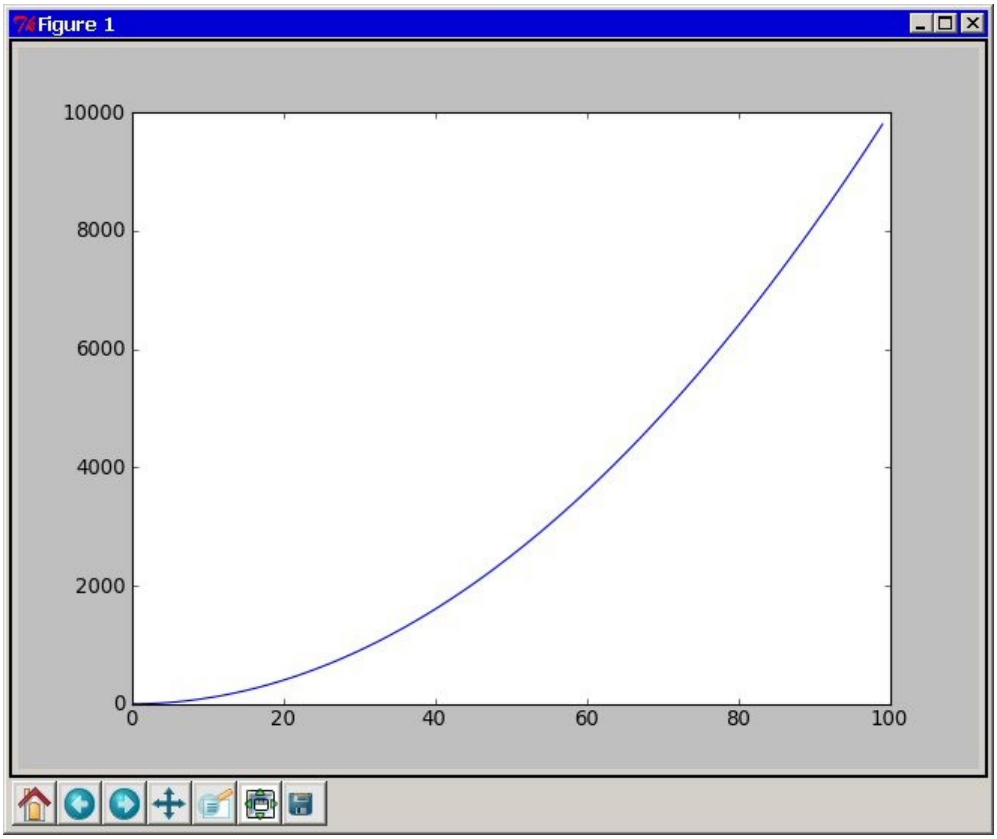
## Plotting with matplotlib

matplotlib (http://matplotlib.sourceforge.net/) is a python library used to produce publication-quality plots. Check out the gallery page on the matplotlib website for examples of the plots that can be done.

I like matplotlib because its strength is getting you a plot in just a few lines of code. Yes, to tune it just the way you want it, you'll have to add a number of other function calls, but doing the basic plots is easy. In fact, if we have two sequences of data in the x and y arrays, you can be looking at a plot with a total of three lines of code (shown in red in the following listing). Here's an example that plots the squares of the integers:

```
from pylab import *

x = range(100)
y = [i*i for i in x]
plot(x, y)
show()
```

This results in the plot




The `show()` command results in an interactive Tcl/Tk plot of the graph. You can put the mouse over points on the graph and read off the coordinates. If you instead had wanted the output in a file, you'd replace the `show()` command with something like `savefig("filename.png", dpi=150)`. The `dpi` parameter determines the size of the bitmap. The file name's extension determines the type of file:

| Extension | File type |
|---|---|
| png | Portable network graphics |
| ps | PostScript |
| pdf | Adobe's Portable Document Format |
| svg | Scalable vector graphics |

If you use an unrecognized extension, you'll get the following list of acceptable ones:

```
Supported formats: emf, eps, pdf, png, ps, raw, rgba, svg, svgz.
```

Where matplotlib is useful is in plotting numpy arrays. The `plot()` command's data arguments can be lists and tuples as well as numpy vectors. Thus, the guts of the previous example could be replaced by the commands

```
from pylab import *

x = arange(0, 100, 1)
plot(x, x*x)
show()
```

The ability to use numpy's expressions can shorten your programs.

I don't have space for much more of matplotlib's details, so I'll refer you to its documentation. However, here's a list of some of the types of graphs it can produce

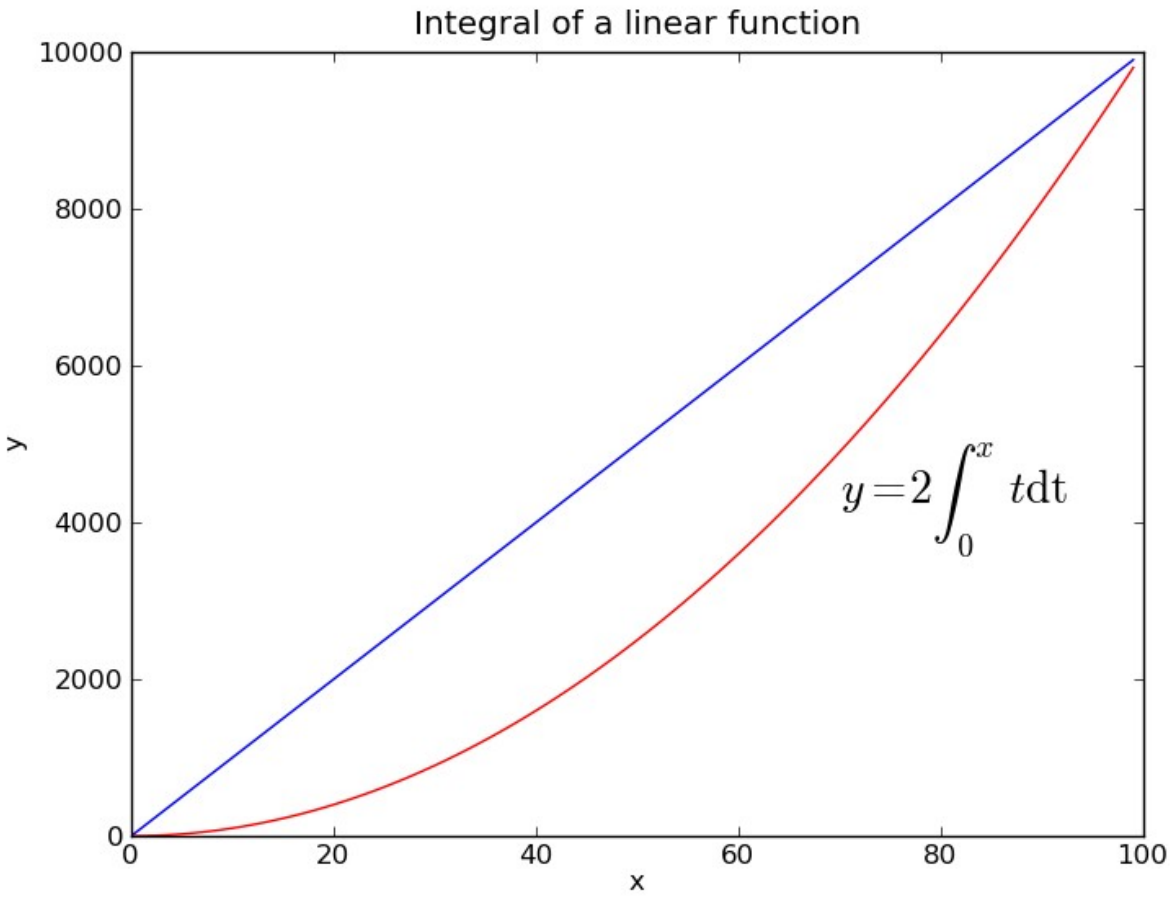
- Regular plots
- Semilog and log-log plots

- Histograms
- Autocorrelations
- Bar charts
- Box and whisker plots (a favorite of mine for comparing data sets)
- Contour plots
- Pie charts
- Polar coordinate plots
- Scatter plots
- Power spectral density plots
- Direction field plots

You can place text anywhere on a plot. You can place mathematical strings on the page too using a TeX-like syntax:

```
from pylab import *

x = arange(0, 100, 1)
plot(x, 100*x)
plot(x, x*x, "r")
xlabel("x")
ylabel("y")
title("Integral of a linear function")
text(70, 3500, r"$y = 2\int_0^x t \rm{d}t$", fontsize=20)
show()
```

produced the following plot




Note that the two `plot()` commands resulted in two different curves (I had to multiply the first set of

ordinates by 100 to put them on the graph in a reasonable fashion). I wanted the second one in red, so I included a color specifier `"r"` for red.

One of the things I use numpy and matplotlib for frequently is doing Monte Carlo calculations. This is a powerful method of modeling systems with stochastic behavior. For an example, see the **Monte Carlo modeling of measurement uncertainty propagation** on page 35.

## Scientific computing with scipy

SciPy is a collection of tools useful in scientific computing and heavily utilizes numpy arrays. Browse http://www.scipy.org/; you may decide that you also want its tools in your toolkit.

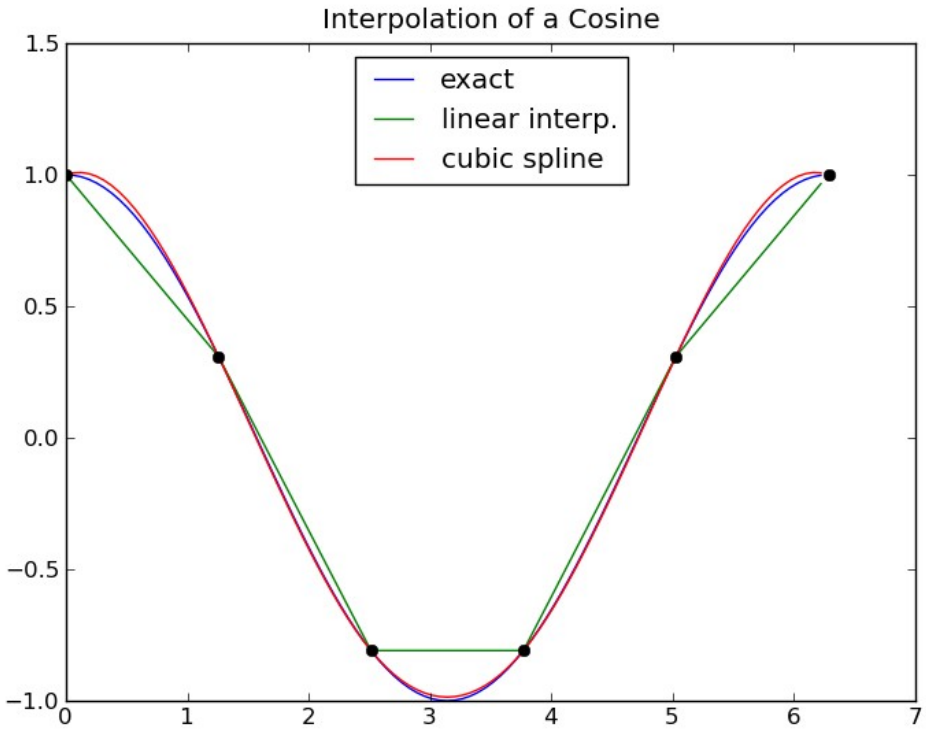
SciPy's documentation can sometimes be a bit confusing and lacks a good hierarchically-organized table of contents. This means you'll have to browse a while to find what you want. Still, there are hundreds of different functions and they can save you a lot of time by not having to write something yourself.

I'll look at two examples here. The first example involves interpolation. We'll plot a cosine, then sample that cosine at five points and generate interpolation functions for the cosine, based on these five points. The two types of interpolation are linear and cubic spline. The plot gives you an idea for how well the interpolation functions can work. Here's the code:

```
from __future__ import division
from pylab import *
from scipy.interpolate import interp1d

twopi = 2*pi

# Generate an exact cosine
n = 100
x = arange(0, twopi, twopi/n)
y_exact = cos(x)

# Now generate a cosine, but only at a small number of points.  The extra
# point is to avoid a value we need to interpolate that is above the
# interpolation range.
n = 5
delta = twopi/n
x1 = arange(0, 2*pi + delta, delta)
y_small = cos(x1)

# Create a linear interpolator function for this small number of points
f1 = interp1d(x1, y_small)

# Create another interpolator function using cubic splines
f2 = interp1d(x1, y_small, kind="cubic")

# Compare the exact vs. interpolated on the plot
plot(x, y_exact, label="exact")
plot(x, f1(x), label="linear interp.")
plot(x, f2(x), "-", label="cubic spline")
# Plot the points used for interpolation as black circles
plot(x1, y_small, "ko")
title("Interpolation of a Cosine")
legend(loc="upper center")
```

The key code is in red. The `interp1d` function returns a function object that can be called with a scalar or numpy vector value as an argument and returns, respectively, a scalar or a vector.

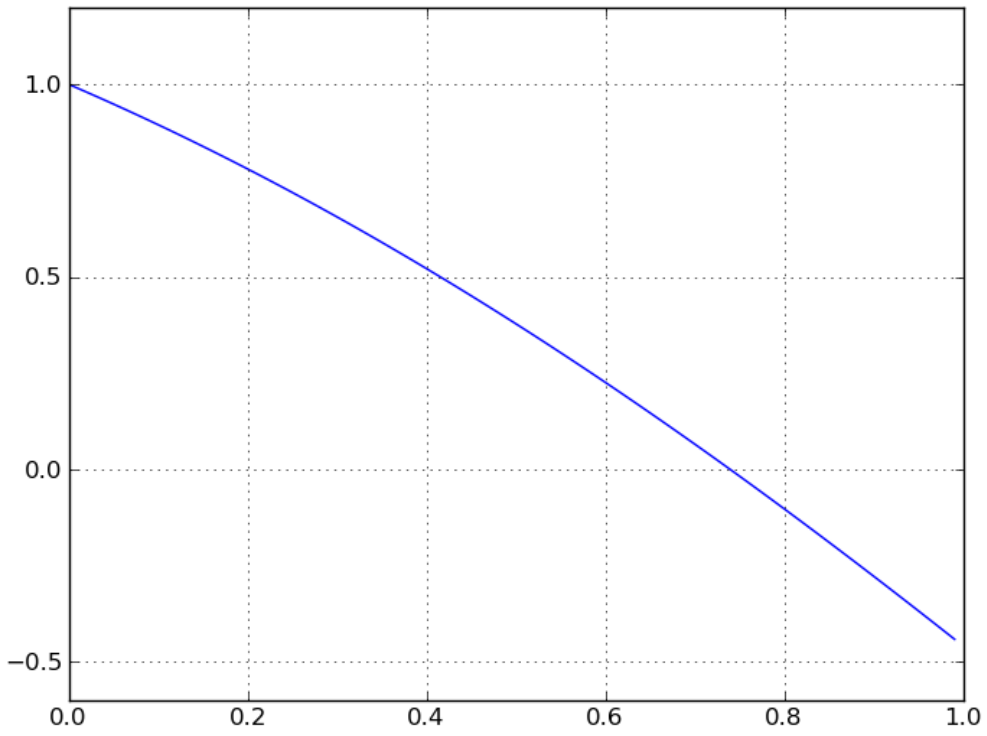
This resulted in the following plot:




If you have lots of points, linear interpolation may be adequate for your needs. With fewer points, cubic spline interpolation will likely work a lot better, shown by how close the red curve is to the blue curve. These interpolation functions will be quite handy when you have a set of experimental data that you'd like to use as a function; this can be much less work and less messy than fitting some curve to the experimental data. However, you cannot extrapolate with these functions as you can a fitted function.

The second example involves a common numerical problem: root finding. SciPy's root finding methods are: Brent's method, Ridders' method, bisection, and Newton-Raphson. Suppose we want to find the first nonzero root of cos(x) - x = 0. A quick plot estimates the root:

```
from pylab import *
x = arange(0, 1, 0.01)
plot(x, cos(x) - x)
grid()
show()
```




We see the root is around 0.73 from the graph. We use Ridders' method to find the root; the 0.7 and 0.8 are required because this routine (as do Brent and bisection) require you to bracket the root.

```
from pylab import *
from scipy.optimize import ridder

print ridder(lambda x: cos(x) - x, 0.7, 0.8)
```

This is perhaps an occasion where the use of an anonymous function isn't too objectionable. The routine prints out 0.739085133216, which you can find with your calculator by iteration (put it into radian mode, enter 0.7, then repeatedly press the cosine key until it converges).

When you don't have any idea where the roots are, one strategy is to do a search. Here's a function which divides an interval up into n subintervals and sees if there's a root in the subinterval.

```
def SearchIntervalForRoots(f, n, x1, x2):
    '''Given a function f of one variable, divide the interval [x1, x2]
    into n subintervals and determine if the function crosses the x axis in
    each subinterval.  Return a tuple of the intervals where there is a
    zero crossing.
    '''
    assert f and n > 0 and x1 < x2
    x0, y0, delta, intervals = x1, f(x1), (x2 - x1)/(n + 1.), []
    for i in xrange(1, n + 1):
        x = x1 + i*delta
        y = f(x)
        if y0*y < 0:
            intervals.append((x0, x))
        x0, y0 = x, y
    return tuple(intervals)
```

The function returns any intervals where the function is of opposite sign. You can then use Brent, Ridders' method, or bisection to further polish the root. Of course, the danger is that the intervals

can be too coarse -- then you can miss some roots.

## Symbolic computation with SymPy

SymPy (http://code.google.com/p/sympy/) is a pure-python project intended to provide a computer algebra system. While it doesn't have the power of commercial systems like Maple or Mathematica, it can still do respectable tasks. You can peruse the documentation at sympy's website to learn more. Here, I'll give an example of a handy capability of sympy. We'll use sympy to evaluate an algebraic expression, then plot the resulting expression to see what it looks like.

The task is to plot Wilkinson's pathological polynomial (you'll have to read [acton] to find out why it's pathological). This polynomial has roots from -1 to -20 inclusive.

```
 1 from pylab import *
 2 import sympy as S
 3 from operator import mul
 4
 5 x = S.Symbol('x')
 6 # Construct Wilkinson's pathological polynomial with roots from -1 to -20
 7 # inclusive (see [acton], pg 201).
 8 #
 9 # Create a list of the terms:  [x + 1, x + 2, ..., x + 20]
10 p = [(x + i) for i in range(1, 21)]
11 # Calculate the product of all the terms.
12 p = reduce(mul, p)
13 # Algebraically expand it and print it out to see the coefficients
14 print p.expand()
15 # Create a numpy array of values and call it x
16 n, delta = 1000, 0.1
17 x = arange(-n, n, delta)
18 # Evaluate x in the polynomial
19 s = "y = " + str(p)
20 exec s
21 plot(x, y)
22 show()
```

Line 5 constructs a sympy variable. Because the variable `x` has number semantics, we just use python arithmetic to construct the polynomial. Of course, what's really going on behind the scenes is that python calls the sympy `Symbol` object's methods for addition and multiplication, letting the object handle the math semantics. Line 10 constructs a list of terms of the form (`x` plus an integer). Line 12 uses the functional programming command `reduce()` to apply the multiplication operator to the list; the result is

$$p = \prod_{i=1}^{20} (x+i)$$

Note the power of the notation -- the two lines 10 and 12 would normally be put on one line[19]; you could then calculate this product for any number of terms desired. Line 14 causes the product to be algebraically expanded and printed out:

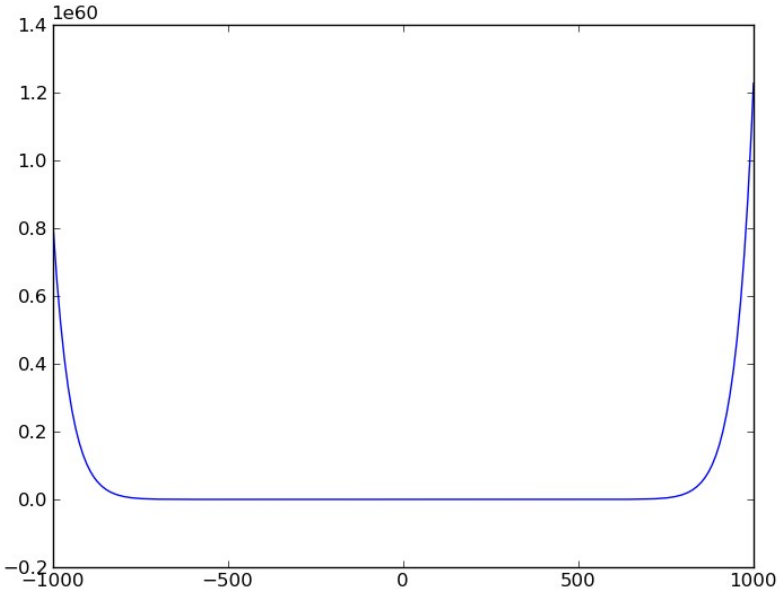
```
2432902008176640000 + 8752948036761600000*x + 13803759753640704000*x**2 +
12870931245150988800*x**3 + 8037811822645051776*x**4 + 3599979517947607200*x**5
+ 1206647803780373360*x**6 + 311333643161390640*x**7 + 63030812099294896*x**8 +
10142299865511450*x**9 + 1307535010540395*x**10 + 135585182899530*x**11 +
11310276995381*x**12 + 756111184500*x**13 + 40171771630*x**14 +
1672280820*x**15 + 53327946*x**16 + 1256850*x**17 + 20615*x**18 + 210*x**19 +
x**20
```

Those coefficients are pretty large -- on the order of $10^{19}$.

In line 17, we create a numpy array `x` of values at which to evaluate the polynomial. Line 19 creates

19 This is an example where using the functional programming keyword reduce is substantially shorter than using other python constructs.

a string of the form `"y = (x + 1)*(x + 2)*..."`, then the next line evaluates it using python's `exec` command. Since `x` is a numpy array that is in scope, the normal arithmetic is done as indicated on the array's elements and `y` holds the result array. We then just plot `y` as a function of `x`. Here's the graph:




The ordinates are way too large for you to see the graph crossing the axis at the roots, even if you reduce the plot interval a lot. However, the objective wasn't to investigate the equation, but rather show you how you could perform symbolic manipulations using sympy, then convert the results into numerical expressions that can be used with numpy arrays and matplotlib for plotting.

Since sympy can calculate the Taylor series for functions, this can help you quickly evaluate whether a particular truncated Taylor series is a good enough approximation for you.

## mpmath

mpmath (http://code.google.com/p/mpmath/) is a pure-python library for doing arbitrary precision real and complex calculations. It also provides a large library of special functions and other useful stuff. Most of mpmath's functions are defined over the complex plane.

For elementary functions with arbitrary arithmetic, one can often get by with the `decimal` module (see the module's documentation for some code for the trig functions using the series expansions). However, when you want to use common "higher" special functions, you can turn to a library like mpmath. One of the uses for an arbitrary precision package is to find out whether your calculations with python's floats are suffering from roundoff error.

Besides real and complex numbers, mpmath provides interval numbers that are useful for studying roundoff error. Use them to do a calculation, then examine how much spread there is in the answer. If it's too much, you may be able to get what you need by increasing the number of decimal digits used in the calculations. Or, you may need to find a better version of your equation that's not so sensitive to roundoff (usually a better thing to do than just increasing the precision).

I give an example of the use of mpmath's interval numbers in the section **Monte Carlo modeling of measurement uncertainty propagation** on page 35.

# Other stuff

This section is a collection of miscellaneous things.

If you need to control one or more serial ports using python, you can download the pySerial package. It works on Windows and Linux. I've used it with the inexpensive USB to serial adapters to control electronic instrumentation. http://code.google.com/p/rs22812/ contains python code that lets you talk to a Radio Shack 22-812 digital multimeter over its serial port.

Python can talk to USB and GPIB instrumentation using the PyVISA library (it also supports serial communications). You'll need a VISA interface that comes with adapters from e.g. National Instruments or Agilent.

The [res] entry in the References section gives some web-based references. The Python Cheese Shop has quite a few packages, although they're of varying quality. There used to be a web site named Vaults of Parnassus that was a good collection of python tools, but it has apparently disappeared. You can of course also search sites like freshmeat and SourceForge for various python solutions. When I'm looking for a solution to a particular problem, I often can find it by a Google search starting with "python", followed by keywords describing the problem.

# Examples

This section discusses some of the programs that come in the package. These are intended to both illustrate some features of python and its libraries as well as be useful stand-alone programs.

## Resistor combinations

The `res.py` script demonstrates a brute-force solution to a problem. As computer performance has increased over the last few decades (and memory has gotten cheaper), a distinct "sea change" has occurred in programming. It used to be that one would spend lots of time figuring out the right algorithm and data structures to solve a problem -- mainly because if the wrong choices were made, your program might not finish in a reasonable time. Today, speed and memory are often of secondary importance compared to the human resources needed to attack the job. This means a brute-force solution, while not elegant, may (I repeat, may) be an acceptable solution. I've found from experience that it always makes sense to first code such problems in python first. If the performance isn't acceptable, then a C++ solution can be tried; it will typically get from 4 to 10 times better performance.

A person working with electronics often needs a particular resistor value, but manufacturers don't make a resistor in that value (or they do, but the person doesn't have one handy). A common technique is then to make a substitute for that resistor by combining two on-hand resistors in series or parallel. Finding the resistor values to do this is a perfect task for a computer. The script uses the `itertools` module's `combination` iterator to construct all combinations of a given set of resistance values, both parallel and series.

As usual in programs that have powerful libraries available, most of the code is involved in dealing with the human: getting input, checking values, and formatting output. The work of the script is done in the `Find()` function. It uses small anonymous functions to calculate the series and parallel values of two resistors. The core work is the `for` loop, which loops on all combinations of the resistors. The `GetResistors()` generator is used to provide all resistance values given the EIA mantissa values and the allowed powers of 10. This is very "pythonic", as it's terribly easy to see exactly what is going on. The generator only produces one value at a time, so there's no large array of values in memory.

Note that the set of resistor values needed to have each resistance value duplicated once to allow for the case where we could make the desired resistance from two resistors of the same value. This was easy to do by just using the `yield` keyword on each value twice.

A more "pythonic" solution would be to put the `Series` and `Parallel` functions into an array and loop on them too, but this would also require the "`series`" and "`parallel`" strings to be included; I felt this would be a little too hard to read, so I just used the easy approach of mostly duplicating code.

Note also that a set is used as the data structure to contain the results. This is done because sets do not allow duplicate values. A dictionary could have been used for this (and was the common way before sets were added to python), but isn't quite as a "pythonic" solution because additional processing would be needed on the dictionary's contents. After you've worked with python for a while, you may sense that the fundamental containers (lists, tuples, dictionaries, sets, and strings) were very well chosen, as their capabilities are nicely orthogonal.

## Finding all primes less than a number n

Another indication of the sea change in computing in recent decades is that in the old days, people would use printed lists of prime numbers in their work. Today, it is typically faster to compute these lists of primes when needed, at least when n isn't too large. Here's an elegant program from the web:

```
    def Primes(n):
        " Returns a list of primes < n."
1       sieve = [True] * (n//2)
2       for i in xrange(3, int(n**0.5) + 1 ,2):
3           if sieve[i//2]:
4               sieve[i*i//2::i] = [False] * ((n-i*i-1)//(2*i)+1)
5       return [2] + [2*i + 1 for i in xrange(1, n//2) if sieve[i]]
```

This is an elegant and readable implementation of the Sieve of Eratosthenes. It's also speedy for an interpreted language -- on my computer, it will find all the primes less than 10 million in about a second. Let's look at how it works.

Line 1 uses a python shorthand of constructing a duplicate of a sequence: multiplying a sequence by an integer k returns k copies of that sequence. Thus, `"abc"*2` returns `"abcabc"` and `["a", "b", "c"]*2` returns `["a", "b", "c", "a", "b", "c"]`. This means line 1 fills `sieve` with `n//2` True values (i.e., `sieve` is an array of Booleans). `n//2` signifies integer division; thus, `3//2` is `1`. These Boolean values will represent the odd integers from 3 to `n//2`. This cuts down on storage space, since even integers can't be primes.

The `for` loop in line 2 goes from 3 to the largest integer less than or equal to the square root of n. This is the correct limit because n cannot have a factor greater than $\sqrt{n}$.

The heart of the algorithm is in line 4. If the `i//2`-th Boolean is True (meaning `2*i + 1` is a candidate prime), this line then sets every multiple of this candidate to false, since these multiples obviously can't be prime. This is done using the extended slice notation. The `[i*i//2::i]` means to start at index `i*i//2` and increment by i. Thus, if i is 3, this means to go from index 4 on up in steps of 3 and set each element to False. The right hand side constructs a Boolean array of the correct size to substitute for the slice's elements.

Let's instrument the code a bit to help us see what's going on. The following listing shows some added code that shows how the sieve's contents change with each iteration:

```
1 def Primes(n, show=False):
2     ''' Returns a list of primes < n.
3     If show is True, print out the sieve's contents for each iteration.
4     '''
5     def Show(count, sieve):
6         if show:
7             # Make a string of 0's and 1's
8             s = str([0 + i for i in sieve])
9             if not count:
```

```
10                 # Print a ruler
11                 print " "*4, ("    .    |")*(len(sieve)//10)
12             print "%3d " % count, s[1:-1].replace(",", "").replace(" ",
13                 "").replace("0", " ").replace("1", "x")
14     sieve = [True] * (n//2)
15     if show:
16         Show(0, sieve)
17     for i in xrange(3, int(n**0.5) + 1 ,2):
18         if sieve[i//2]:
19             sieve[i*i//2::i] = [False] * ((n-i*i-1)//(2*i)+1)
20             if show:
21                 Show(i, sieve)
22     return [2] + [2*i + 1 for i in xrange(1, n//2) if sieve[i]]
23
24 print ' '.join(str(i) for i in Primes(int(100), True))
```

This results in

```
        .    |    .    |    .    |    .    |    .    |
  0  xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
  3  xxxx xx xx xx xx xx xx xx xx xx xx xx xx xx xx xx
  5  xxxx xx xx x  xx  x xx xx x  xx  x xx xx x  xx  x
  7  xxxx xx xx x  xx  x xx x  x  xx  x xx  x x  x   x
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

The x's in the table show which odd numbers are still known to be prime. The first x represent 1 and each subsequent position represents the next odd number. Thus, you can read off the first few primes as 1, 3, 5, 7, 11, and 13. (Of course, 1 isn't considered prime, but it's part of the array to ease the bookkeeping.)

Now, back to the example we were discussing. When `i` is 3, the first pass through the loop, all the elements in the `sieve` list are True. Thus, line 18's conditional (in the immediately previous listing) is true, so line 19 is executed. The extended slice is `[4::3]`, which means to start at index 4 and go up in steps of 3. You can see in the results for `i == 3` the locations that were set to False. If you count the number of blank spaces, you'll get 16. The multiplier on the right-hand-side of line 19 is

$$\left\lfloor \frac{n-i^2-1}{2\mathrm{i}} \right\rfloor + 1 = \left\lfloor \frac{100-3^2-1}{2(3)} \right\rfloor + 1 = \left\lfloor \frac{90}{6} \right\rfloor + 1 = 15+1 = 16$$

which agrees with the count. Here, the half-square brackets mean the "integer part of".

Note you don't see a row printed out with `i == 9` -- the 5th element of the sieve was set to False for `i == 3`, so the conditional is skipped.

## frange: floating point analog of range()

`range()` produces an arithmetic sequence and `xrange()` is the same thing but in a generator form[20]. The full form is `range(m, n, inc)`, where `m` is the starting value of the sequence, `n` is the stopping value, and `inc` is the increment. You thus get a sequence of integers `m`, `m + inc`, `m + 2*inc`, etc., up to but not including `n`. This behavior mirrors the behavior of python's slice notation.

Unfortunately, these two functions only support integer arguments. This is a disappointment, as it would be nice if the built-in also supported floating point sequences. However, as soon as you write one, you find that it's not obvious how to write one that behaves properly. You'll find numerous implementations on the web, but they are implemented with the floating point arithmetic built into python and suffer from the flaws of floating point arithmetic. They will usually fail to give the proper sequence for some choice of parameters, failing by giving one number too many some times.

A decade or more ago I also implemented such a python float-based `frange()` and suffered with its failures. Then I wrote one that is more properly implemented to reduce the failure of occasionally

20 In python 3, range behaves like xrange and xrange is eliminated.

providing one too many numbers in the sequence. It's in the `frange.py` module included in the zip file. It has a couple of things to note.

First, it illustrates the power of writing things such that different math implementations can be used. By default, the `frange()` method uses python's `decimal` module to get proper sequences. But different numerical types can be substituted. A useful example in the module's tests is to generate sequences of rational numbers using pythons `fractions` module. In fact, any numerical object that has the proper arithmetic semantics and can be converted to an integer with the `int()` function can be used to generate sequences.

Second, the module demonstrates the use of both the `unittest` and `doctest` modules for testing. They work fine side-by-side. I deliberately used `doctest` to allow the `frange()` function's docstring to contain information on how to use the function.

## Monte Carlo modeling of measurement uncertainty propagation

numpy and matplotlib together provide a powerful team to help with Monte Carlo calculations. I'll illustrate with an example of calculating the propagation of uncertainty in physical measurements. Most college science students are probably exposed to the classical analytical formulas [unc] for what used to be called error propagation (but is now labeled with uncertainly propagation, definitely a better name). However, using the formulas in real-world problems often leads to too much algebraic complexity because of the complexity of the partial derivatives. I've found in practice that it's much easier to go straight to a Monte Carlo calculation of the results. In fact, the method is so simple that it can be shown to high school students who can use it profitably and be doing analyses that would befuddle a PhD statistician if he had to do them analytically.

The reason the Monte Carlo method is powerful is because it can easily be used to simulate situations that are beyond the capabilities of the analytical method. Since we usually only need two or three significant figures at best in an error estimate, Monte Carlo calculations are quite possible.

We'll use the example of a voltage divider. Suppose a voltage source V is put across a voltage divider made up of two resistors R1 and R2. The output is taken from across resistor R2. The output voltage $V_o$ is

$$V_o = \frac{R2}{R1+R2} V$$

Suppose we are making a high precision voltage divider and we want to make an uncertainty statement about the (stochastic) variable $V_o$ in terms of the stochastic behavior of the independent variables R1, R2, and V. The independent variables are stochastic because they are measured with an instrument that has measurement uncertainty. We want to combine the measurement uncertainties in the independent variables to help us make a statement about the measurement uncertainty of $V_o$.

Let's assume that our measured values are: R1 = 1205 $\Omega$, R2 = 1477 $\Omega$, and V = 3.91 volts. We check that our instrument is within its calibration period. The manufacturer of our instrument states that the uncertainty of a voltage measurement is ±0.5% and the uncertainty of a resistance measurement is ±1%.

Unfortunately, manufacturers don't specify the uncertainty distribution, so one is usually forced to assume the true value of the measurement is uniformly distributed in the uncertainty interval. Thus, we are led to the following algorithm to calculate the uncertainty in $V_o$:

```
count = 0
Create array to store Vo values
while count < number of times to repeat:
      Generate R1 from U(1205 ± 1%)
      Generate R2 from U(1477 ± 1%)
      Generate V from U(3.91 ± 0.5%)
      Calculate Vo = R2/(R1 + R2)*V
      Save Vo in the Vo array
```

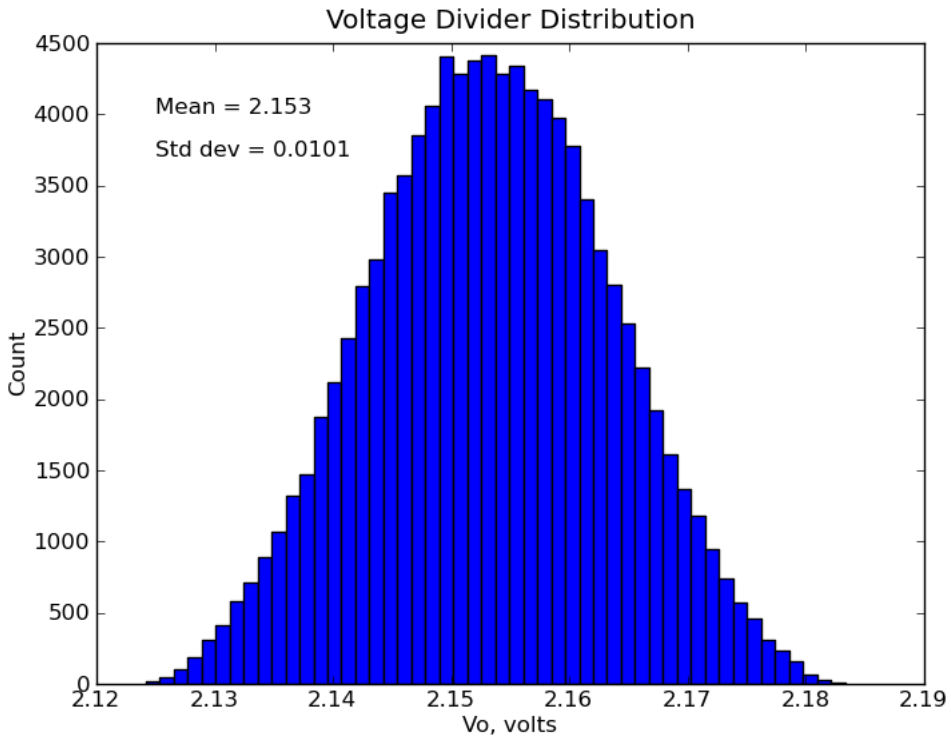
### Plot a histogram of Vo

Fortunately, this translates easily into python:

```
from pylab import *
import random

N = int(1e5)   # Number of times to repeat "experiment"
R1, R2, V = 1205, 1477, 3.91
Vo = []
for i in xrange(N):
    r1 = random.uniform(R1*(1 - 0.01), R1*(1 + 0.01))
    r2 = random.uniform(R2*(1 - 0.01), R2*(1 + 0.01))
    v  = random.uniform(V*(1 - 0.005), V*(1 + 0.005))
    vo = r2/(r1 + r2)*v
    Vo.append(vo)
hist(Vo, bins=50)
title("Voltage Divider Distribution")
xlabel("Vo, volts")
ylabel("Count")
text(1.725, 4000, "Mean = %.4g" % average(Vo))
text(1.725, 3700, "Std dev = %.3g" % sqrt(cov(Vo)))
```

The heart of the calculation is the line in red. This one line performs the Monte Carlo calculation once the variables' arrays have been filled with appropriate values. I hope you agree that this is simple and powerful -- and quite accessible to folks who don't have lots of statistical training.

The result is




Note that the distribution of the dependent variable looks much like a normal distribution. This is a common behavior and you'll see it often when you experiment with different distributions. It's not quite the Central Limit Theorem or its more general versions, but it sure does rhyme. I took my statistics class a few decades ago and don't remember if there's a theorem covering this, so if someone knows if there is some theorem that describes this behavior, please let me know. The closest I know of is one of Cramér's Theorem given in Ku's article (see [unc]), but that theorem (and the Central Limit Theorem) typically describe what happens to statistics, not the distributions.

One has to choose how to report this uncertainty. Since the independent variables were given in percentage of reading with the uncertainty extending uniformly over the whole band, we could approximate this by saying the dependent variable will extend ±3 standard deviations. We could then turn this into a percentage: 2.153 volts ± 1.4%.

One of the powerful features of using python, numpy, and matplotlib for Monte Carlo studies like this is that it's not a lot of extra work to prepare distributions that are not the usual elementary distributions. For example, in high tech industries, one can see manufacturing processes that do acceptance sampling; these typically result in truncated distributions. In addition, physical process changes/shifts can cause bimodal or multimodal part distributions. These types of distributions can be constructed relatively easily using numpy, so doing simulations with them is straightforward -- and these simulations can then provide you with estimates of what will really happen.

Another way of estimating the propagation of uncertainty is to use interval arithmetic to perform the calculation. This is a more conservative technique than needed because the typical uncertainty distributions are peaked. However, it does provide guaranteed bounds inside of which the dependent variable must lie. Here's the calculation using interval numbers[21]:

```
from mpmath import *
R1, R2, V = 1205, 1477, 3.91
r1 = mpi(R1*0.99, R1*1.01)
r2 = mpi(R2*0.99, R2*1.01)
v  = mpi(V*0.995, V*1.005)
print r2/(r1 + r2)*v
```

Here are the results:

```
[2.1000777842381546101, 2.2077542025399412573]
```

This would turn into a percentage specification of 2.154 ± 2.5%. Note how the uncertainty interval is substantially larger than the one gotten via the Monte Carlo model.

## Source code control

This section doesn't give an example of python solving a problem, but it discusses something that might be important to you.

The process of working on source code usually requires some control over the files and information making up the application. Beginners often start out with file naming conventions, multiple directories, and other techniques to keep revisions straight. This can work, but it's clumsy and error-prone. When you suddenly have to start working on the same information with one or more other people, it can quickly turn into a nightmare.

This is an old and familiar problem to software developers and they have generated many tools over the years for these problems. First came systems like SCCS and RCS, then came server-based systems like CVS and Subversion. The latest tools are distributed version control tools like git, Mercurial, and Bazaar[22]. Interestingly, the last two are implemented in python. I've chosen Mercurial as my version control tool, although I could be happy with either git or Bazaar too. I'd like to give you an overview for what real-world use of such a tool looks like. I can only scratch the surface here; see [hg] for more details.

There exist GUI-based interfaces to Mercurial (TortoiseHg is one for Windows), but I suspect most people interact with Mercurial using the command line. I'm going to illustrate the use of Mercurial by simulating the creation and development of a simple python project. Assume python and Mercurial are installed and available at the command line by the commands `python` and `hg`, respectively.

21 The evolution of mpmath seems to be de-emphasizing the use of interval numbers. I'm using an older version (0.12); this example may or may not work with the current version.

22 There are also many commercially-available packages besides these open source projects; see http://en.wikipedia.org/wiki/Version_control and http://en.wikipedia.org/wiki/Comparison_of_revision_control_software.

First, we create a new project directory

```
mkdir new_project
cd new_project
```

Next, we create a new Mercurial repository in this directory (it will be in the hidden directory `.hg`). This repository will let us track the revisions of any of the files at and below this directory. Note you have to add the files to the repository by telling Mercurial to track them; they are not tracked automatically for you.

```
hg init
```

Suppose we now edit a new file named `project1.py`. We want to put this file under version control, so we execute

```
hg add project1.py
```

We also add a file `test.data` that supplies test data for the self-tests written in the project1.py file:

```
hg add test.data
```

If we execute the command `hg status`, we see

```
A project1.py
A test.data
```

Mercurial is telling us that these two files have been added to the repository, but haven't been committed yet. Let's perform a commit, which stores the files' data in the Mercurial database in the fashion of a database transaction

```
hg commit -m "Initial checkin"
```

We can see the history of checkins by:

```
$ hg log
changeset:   0:d888b5013b06
tag:         tip
user:        donp
date:        Sat Nov 06 11:43:01 2010 -0600
summary:     Initial checkin
```

The state of the files in the repository is codified by the hex number `d888b5013b06`, which labels the changeset (set of changes leading to this database state from the prior changeset(s)). We can return to this version of the files at any time we want by using this number (or the integer to the left of the colon).

Then the process of software development is (usually) a sequence of changes, additions, and deletions applied to this project. You do a `commit` whenever you feel it might be important to return to a certain point in development (you'll learn through experience that the best rule is "commit often"). I'll stop here, but there's a lot more power to this tool.

If you type `hg` by itself at the command line, you get the help statement

```
Mercurial Distributed SCM

basic commands:

 add        add the specified files on the next commit
 annotate   show changeset information by line for each file
 clone      make a copy of an existing repository
 commit     commit the specified files or all outstanding changes
 diff       diff repository (or selected files)
 export     dump the header and diffs for one or more changesets
 forget     forget the specified files on the next commit
 init       create a new repository in the given directory
```

```
 log         show revision history of entire repository or files
 merge       merge working directory with another revision
 pull        pull changes from the specified source
 push        push changes to the specified destination
 remove      remove the specified files on the next commit
 serve       export the repository via HTTP
 status      show changed files in the working directory
 summary     summarize working directory state
 update      update working directory

use "hg help" for the full list of commands or "hg -v" for details
```

You can get help on an individual command like the `status` command by `hg help status`. Most revision control systems work in a very similar fashion, although of course the commands differ.

### *Revision control system architectures*

There have been three common architectures for revision control systems over the last few decades. First is what I will call workstation-based solutions like SCCS and RCS. These are file-oriented architectures that work on one machine, usually a UNIX-type machine when multiple people worked on a project (e.g., each person logged into the machine and worked on a project's files). By "file-oriented", I mean the revision control system's operation was focused on files, not groups of files. This worked fine for relatively small projects, but becomes cumbersome when the number of files grows and spreads out over many directories. I used RCS as my personal version control tool for about 20 years and, within its limits, it worked just fine.

The next major architecture was server-based (such as CVS and Subversion). The version control database is stored on a central server; workstations connect to the server to get and store files. The server acts as a gatekeeper to validate the incoming connection's credentials. An advantage is that the version information is in one place and is easy to back up. A disadvantage is that if the server goes down, there can be a big work stoppage.

The latest popular architecture is the distributed revision control tool (such as git, Mercurial, or Bazaar). The architectures supports both the central server model of operation as well as having the whole version control database can be stored on individual workstations. This gives flexibility. In addition, this distributed architecture has functionality that incorporates some of the latest ideas in revision control research, giving the experienced user more power.

Both the server-based and distributed architectures can allow the user to view the set of files as a whole and operate on them as a group, rather than having to do things a file at a time. This is significant on larger projects, which can have hundreds of thousands to millions of files.

## References


[acton] F. Acton, *Numerical Methods that Work*, The Mathematical Association of America, 2nd printing, 1990 (first edition was in 1970), ISBN 0-88385-450-3.

[Beazley] David Beazley, *Python Essential Reference*, 2000, New Riders Publishing, ISBN 0-7357-0901-7. This is a small book based on python version 1.5.2 and is excellent because of its brevity and coverage. It's now in its fourth edition, but is no longer a small book.

[ds] At some point in their career, I feel every scientist or engineer should take a Data Structures and Algorithms course, which is a standard course in any undergraduate computer science curriculum. This course will do a lot to help you better understand the process of designing and implementing your programs. It will better prepare you to read material on software design and engineering. Most importantly, it will make you thoughtful about what data structures and algorithms you use in your own programs;

these choices typically have much more impact on performance than what language, compiler, or optimizations you use. If you don't have time to take a course, find a textbook (there are probably hundreds of them) and teach yourself. Find a computer science graduate coworker or friend who can help you over any rough spots.

[eckel] B. Eckel, Thinking in Python, http://www.mindview.net/Books/TIPython. The first chapter is a brief introduction to python for experienced programmers.

[fp] Functional programming references:

http://docs.python.org/howto/functional.html

http://www.ibm.com/developerworks/library/l-prog.html

[GoF] Erich Gamma, et. al., *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995, ISBN 0-201-63361-2.

[hg] Besides the Mercurial website, there are other places to learn about Mercurial. If you're a software developer, I recommend http://hginit.com/; if you've never used Subversion before, skip the first panel. There's a tutorial book http://hgbook.red-bean.com/ that's useful. If you want to host an open source project, BitBucket provides useful tools (git and Bazaar have corresponding sites).

The reason I chose Mercurial for my personal version control system over git and Bazaar was because Mercurial seemed to me to have better support for Windows graphical interactions (http://tortoisehg.bitbucket.org/). This was irrelevant for my own purposes, as I prefer command lines, but it would allow me to work with others who might not have or want to develop the necessary command line proficiency.

[knuth] D. Knuth, *The Art of Computer Programming*, volumes 1-3, Addison-Wesley, early 1980's.

[nr] W. A. Press, et. al., *Numerical Recipes in C*, various editions (and languages), early 1990's. While these books are popular, the programming style is poor.

[res] There are many python resources around the web that are worth scouring for possible solutions to your problems. You'll find a lot of them by searching for your problem with the word "python" included in the search. A few places to look are:

Python cheese shop: http://pypi.python.org/pypi?%253Aaction=browse
Popular python recipes: http://code.activestate.com/recipes/langs/python/
Stack overflow: http://stackoverflow.com/
Sourceforge: http://sourceforge.net/

[unc] Practicing scientists and engineers should turn to http://www.bipm.org/en/publications/guides/gum.html *Evaluation of Measurement Data -- Guide to the Expression of Uncertainty in Measurement*, Working Group 1 of the Joint Committee for Guides in Metrology, 2008 (1995 version with minor corrections). This is the standard document for how to express uncertainty in measurement; it has played an important role in filling the need to standardized the reporting and interpretation of uncertainty in physical measurements. Technically, it's longer than it needs to be (over 100 pages), but it was written with care by a committee and they wished to give extra material, examples, and definitions to make sure users got the concepts. The link given is to the page containing the links to the various documents; the actual link to the PDF is http://www.bipm.org/utils/common/documents/jcgm/JCGM_100_2008_E.pdf. The document http://www.bipm.org/utils/common/documents/jcgm/JCGM_101_2008_E.pdf *Propagation of distributions using a Monte Carlo method* is a 90 page supplement to the

first document; it discusses using the Monte Carlo method as an alternative method to the "classical" (analytical) uncertainty propagation methods.

Unfortunately, these documents might be a bit out of the reach of people who haven't had a college-level statistics course. However, if you're doing careful scientific work, you have to master the material in these references to be able to calculate, propagate, and report your measurements in a meaningful fashion. Alas, a lot of the published papers in the literature show complete ignorance of these techniques -- which is sad, because people can't meaningfully build on the paper's results.

A good but dated reference is National Bureau of Standards, *Precision Measurement and Calibration*, H. Ku, editor, Special Publication 300, Volume 1, 1969. Ku's article in this compilation, *Notes on the propagation of error formulas*, is worth consulting for information on the analytical methods. An article that should be read by every science teacher who insists on foisting the flawed significant figures method as an uncertainty propagation tool on innocent and unsuspecting students is D. B. De Lury, *Computations with approximate numbers*, The Mathematics Teacher, Nov. 1958; this is reprinted in the NBS compilation.